SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations

Changan Niu
State Key Laboratory for Novel
Software Technology
Nanjing University
Nanjing, China
nougatca@qq.com

Chuanyi Li
State Key Laboratory for Novel
Software Technology
Nanjing University
Nanjing, China
lcy@nju.edu.cn

Vincent Ng
Human Language Technology
Research Institute
University of Texas at Dallas
Richardson, Texas, USA
vince@hlt.utdallas.edu

Jidong Ge
State Key Laboratory for Novel
Software Technology
Nanjing University
Nanjing, China
gjd@nju.edu.cn

Liguo Huang
Dept. of Computer Science
Southern Methodist University
Dallas, Texas, USA
lghuang@lyle.smu.edu

Bin Luo
State Key Laboratory for Novel
Software Technology
Nanjing University
Nanjing, China
luobin@nju.edu.cn

ABSTRACT

Recent years have seen the successful application of large pre-trained models to code representation learning, resulting in substantial improvements on many code-related downstream tasks. But there are issues surrounding their application to SE tasks. First, the majority of the pre-trained models focus on pre-training only the encoder of the Transformer. For generation tasks that are addressed using models with the encoder-decoder architecture, however, there is no reason why the decoder should be left out during pre-training. Second, many existing pre-trained models, including state-of-the-art models such as T5-learning, simply reuse the pre-training tasks designed for natural languages. Moreover, to learn the natural language description of source code needed eventually for code-related tasks such as code summarization, existing pre-training tasks require a bilingual corpus composed of source code and the associated natural language description, which severely limits the amount of data for pre-training. To this end, we propose SPT-Code, a sequence-to-sequence pre-trained model for source code. In order to pre-train SPT-Code in a sequence-to-sequence manner and address the aforementioned weaknesses associated with existing pre-training tasks, we introduce three pre-training tasks that are specifically designed to enable SPT-Code to learn knowledge of source code, the corresponding code structure, as well as a natural language description of the code without relying on any bilingual corpus, and eventually exploit these three sources of information when it is applied to downstream tasks. Experimental results demonstrate that SPT-Code achieves state-of-the-art performance on five code-related downstream tasks after fine-tuning.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;
• **Computing methodologies → Artificial intelligence**.

KEYWORDS

pre-training, code representation learning, sequence-to-sequence

1 INTRODUCTION

Pre-Training has revolutionized the way computational models are trained in the natural language processing (NLP) community [12, 13, 43, 44]. For a long time, supervised learning has been the most successful natural language learning paradigm. The pioneers of the pre-training idea challenged this view by showing that a vast amount of general knowledge about language, including both linguistic and commonsense knowledge, can be acquired by (pre-)training a model in a *task-agnostic* manner using *self-supervised* learning tasks. Self-supervised learning tasks are NLP tasks for which the label associated with a training instance can be derived automatically from the text itself. Consider, for instance, one of the most well-known self-supervised learning tasks, Masked Language Modeling (MLM) [13]. Given a sequence of word tokens in which a certain percentage of tokens is *masked* randomly, the goal of MLM is to predict the masked tokens. As can be easily imagined, a model for MLM can therefore be trained on instances where each one is composed of a partially masked sequence of word tokens and the associated "class" value is the masked tokens themselves. Because no human annotation is needed, a model can be pre-trained on a very large amount of labeled data can be automatically generated, thereby acquiring a potentially vast amount of knowledge about language. A pre-trained model can then be optimized for a specific

task by fine-tuning its parameters using task-specific labeled data in the standard supervised fashion.

Inspired by the successes of pre-trained models in NLP, a number of pre-trained models for source code have been proposed and applied to a variety of SE tasks including code summarization and code completion, with notable successes [8, 16, 19, 25, 27, 28, 37, 39, 51]. Despite these promising results, there are issues surrounding the application of pre-trained models for source code to SE tasks.

First, the majority of these pre-trained models focus on pre-training only the encoder of the Transformer [8, 16, 19, 27]. This is not ideal, however. For instance, for generation tasks that are addressed using models with the encoder-decoder architecture, there is no reason why the decoder should be left out in the pre-training process. Second, these models have largely assumed as inputs the source code [8, 27, 28, 37, 51] and the associated natural language description [16, 39]. In particular, code structure, which is also crucial to understanding source code, is largely missing from these models. The reason why code structure is left out is that SE researchers have for the most part simply reused the pre-training tasks designed for natural languages when pre-training models (by viewing source code as natural language), but none of these tasks are concerned with learning the structure of a natural language. Third, these pre-training tasks assume the availability of a bilingual corpus, where each method/function (henceforth collectively referred to as method) is "labeled" with the corresponding docstring, when pre-training a model on source code and the associated natural language description [16, 19]. However, such a bilingual corpus tends to be small in size compared to a monolingual (i.e., source code only) corpus, thus severely limiting the amount of data a model can be pre-trained on. In general, we believe the reliance on a bilingual corpus would hinder the development of powerful pre-trained models for source code in the long run, as a key strength of the pre-trained models developed in the NLP community stems from their ability to be trained using self-supervised tasks for which *very large* amounts of labeled data can be automatically generated.

Several attempts have been made to address the three problems mentioned above to different extents. To address the first problem, T5-learning [39] and TreeBERT [25] are proposed, which are sequence-to-sequence (i.e., seq2seq) pre-training models with the encoder-decoder architecture and enables both the encoder and the decoder to be jointly trained. To address the second problem, GraphCodeBERT [19] employs two pre-training tasks specifically designed to acquire structural information. One task involves predicting the edges in the data flow while the other involves predicting the alignment between the nodes in the data flow and the code sequence, respectively. However, as the authors also pointed out, while the data flow captures information that is largely semantic in nature, it does not capture syntactic information (e.g., the syntactic structure encoded in an Abstract Syntax Trees, i.e., AST), which is arguably the most important type structural information about source code that is commonly exploited by SE researchers. TreeBERT [25] employs the set of constituent paths of ASTs as the input of its encoder. However, it only inputs code sequences at the decoder side during pre-training, trying to make the encoder learn lexical and semantic information (both of which can be easily obtained from the code sequences) from the AST (which contains mainly syntactic information) during the pre-training phase. But

it is uncertain whether TreeBERT can extract the complete lexical and semantic information from the AST only by relying on pre-training, thus eliminating the need to input code tokens when fine-tuning. To address the third problem, T5-learning treats code and natural language as two types of independent data instances. While this allows T5-learning to learn from a monolingual rather than bilingual corpus, the connection between a piece of code and the associated natural language description is no longer present in the corpus. Hence, it is no longer clear whether T5-learning can still learn to produce a natural language description of a piece of code. To our knowledge, there has been no attempt to address all three issues in a single model.

In light of the above discussion, we propose SPT-Code, a new pre-trained model for source code. Motivated by T5-learning, SPT-Code is a seq2seq pre-training model, enabling both the encoder and the decoder of Transformer to be jointly pre-trained. Each data instance for SPT-Code is composed of three types of information derived from a method, namely the code sequence, its AST, and the associated natural language description. Note that the incorporation of ASTs allows SPT-Code to exploit structural, specifically syntactic, information. In addition, to obviate the need to learn natural language descriptions from a bilingual corpus, we will simply use the name of the method and the names of the methods that are invoked in this method as a (very succinct) natural language description of the given source code.

We design three code-specific pre-training tasks for SPT-Code, each of which allows SPT-Code to acquire exactly one of the three types of information that comprise a data instance. The first task is a version the well-known Masked Sequence to Sequence (MASS) [49] pre-training task for natural language that we adapt to source code. Specifically, our modified MASS task seeks to acquire knowledge about source code via masking a random fragment of the code tokens. The second task, Code-AST prediction (CAP), is designed to enable the model to gain knowledge of the syntactic structure of a code fragment by predicting whether the given AST is the correct AST for the given piece of code. The final task, Method Name Generation (MNG), is a novel task that involves generating the subtokens of the method name, which we take to be an (extremely concise) natural language description of the method.

After SPT-Code is pre-trained on the CodeSearchNet dataset [24], we fine-tune and evaluate it on five downstream tasks, including code summarization, code completion, bug fixing, code translation, and code search. Experimental results show that SPT-Code achieves state-of-the-art results under virtually all circumstances.

In sum, we make the following contributions:

(1) Propose SPT-Code, a seq2seq pre-trained model for source code that is built with the encoder-decoder architecture and is applicable to both classification and generation tasks.

(2) Extend the input representation of pre-trained models for source code with a simplified and linearized version of ASTs. To our knowledge, we are the first to use both natural language and AST as inputs in pre-training for source code.

(3) Design special input representations and three code-specific seq2seq-based pre-training tasks enabling SPT-Code to be pre-trained without relying on any bilingual corpus or labeled data.

(4) Pre-train SPT-Code on a large unlabeled monolingual (i.e., source code only) dataset across six programming languages, then fine-tune and evaluate it on five downstream code-related tasks, achieving state-of-the-art results on all tasks.

2 RELATED WORK

2.1 Pre-Training Models in NLP and SE

Table 1 presents an overview of the most prominent pre-trained models in NLP and for code. Each model is characterized along four dimensions: (1) Modules: what is being pre-trained (e.g., the encoder, the decoder, or both); (2) Objectives: the pre-training objectives[1] and (3) Inputs: what information the model assumes as input (e.g., natural language, code and structural information of the code).

We can make a few observations from Table 1. First, while the majority of work has focused on pre-training the encoder, the newest models (T5 [45], BART [34], and T5-learning [39], which is modeled after T5) are all seq2seq pre-training models that allow the encoder and the decoder to be jointly trained. Second, except for GraphCodeBERT [19] and TreeBERT [25], all pre-trained models in SE reuse training objectives designed for natural languages, with MLM being the most popular pre-training task. This indicates that the selection of pre-training tasks, which dictates what knowledge will be acquired and exploited by a model, is an area of research that is under-investigated in SE. Finally, while earlier pre-training models in SE assume only source code as input, the later ones all use both code and language.

In our experiments, we will use as our baselines the most recently developed (and also the state-of-the-art) pre-trained models for source code, namely CodeBERT [16], GraphCodeBERT [19], CugLM [37], T5-learning [39] and TreeBERT [25].

2.2 Structural Information of Source Code

Structural information is very important for understanding source code. ASTs are widely used in code related tasks for representing structure information of code (e.g., [20, 32, 52, 57, 62]), which contains abundant syntactic structure information that cannot be expressed by code sequences. An AST should be flattened with linearization methods, e.g., pre-order traversal [57, 62], in-order traversal [52], and Structure-based Traversal (SBT) [20], before being fed to an encoder. code2vec [6], code2seq [4], and SLM [5] use a method that linearizes an AST as a series of "path-contexts" representing two terminal nodes and the path between them. Jiang et al. [25] represent ASTs as the set of paths and then introduce node position embedding to obtain the position of the node in the

AST. Besides, neural networks that take trees as input (e.g., tree-LSTMs [35, 55, 56], RvNNs [63] and GNNs/GCNs [9, 31, 61]) utilize an AST directly instead of flattening it.

There are also approaches taking data flow and control flow extracted from code as structural information, e.g., [19] and [23]. However, these flows do not contain structural information as rich as ASTs [19]. Their advantages over ASTs are that they have a lower demand on hardware and need less training time under the same conditions [19].

Taken together, we choose to use ASTs to represent the structural information in SPT-Code.

3 SPT-CODE

In this section we first introduce the architecture of SPT-Code (Section 3.1). We then describe the model input (Section 3.2) and the pre-training tasks (Section 3.3), which are the key innovations of this paper. Finally, we illustrates how to fine-tune SPT-Code when it is applied to downstream tasks (Section 3.4).

3.1 Model Architecture

Architecturally, SPT-Code is essentially a multi-layer Transformer [54], which is widely used by pre-training models such as BART [34] and T5 [45]. As far as the parameter setting of the encoder and decoder is concerned, SPT-Code follows CodeBERT and GraphCodeBERT: (1) the number of layers (i.e., Transformer blocks) $L = 12$, (2) the size of model $d_{model} = 768$, (3) the dimension of feed forward $d_{ff} = 3072$, (4) the number of self-attention heads $h = 12$, and (4) the dropout rate $p_{dropout} = 0.1$. The total number of parameters is 262M.

To pre-train both classification and generation tasks with an encoder-decoder structure, we adopt the strategy used in BART. In particular, note that the encoder and decoder will continue to learn jointly and collaboratively when pre-trained on classification tasks.

Figure 1 shows how classification and generation tasks are pretrained in SPT-Code. Specifically, to use SPT-Code for classification tasks, the input of the decoder is the same as encoder, except that a special symbol "[SOS]" (indicating the start of the sequence) is added to the front, and another special symbol "[EOS]" (used here as a placeholder for the classification position) is added to the end. The output of the corresponding position of the "[EOS]" is then used for classification. For generation tasks, Figure 1b demonstrates the input and output of the model when translating "A B C D E" to "α β γ δ ε". The special symbol "[EOS]" denotes the end of the sequence, and the process of generation stops when the model outputs this symbol.
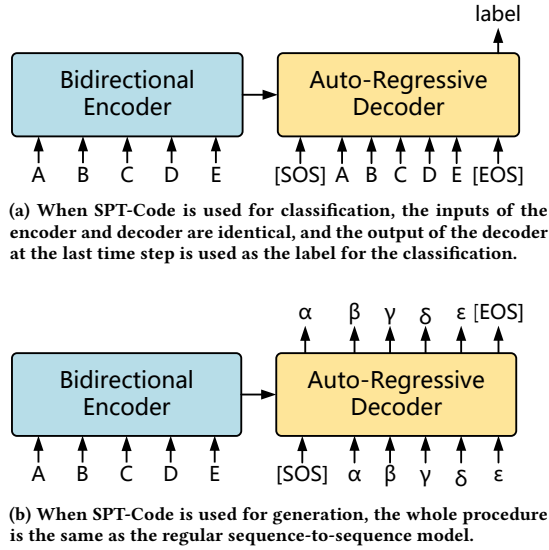
3.2 Model Input

The inputs of the model are three different types of components belonging to a complete *Method* (i.e., it can be invoked by its name), namely, the code tokens, the linearized AST, and the natural language. In this subsection, we will demonstrate this with a real Java method[2] shown in the bottom of Figure 2.

---

[1]Forward LM [28, 43, 44] aims to predict the next word given the preceding words in a sentence. Backward LM [28, 43] aims to predict the previous word given the words that appear after it in a sentence. Masked LM is the masked language modeling task described in Section 1. NSP [13] aims to predict whether the second sentence in a sentence pair should immediately follow the first sentence in the pair. Permutation LM [60] aims to predict a word using a set of context words randomly selected via the attention mask mechanism. RTD aims to predict which token in the input has been replaced. EP [19] involves masking 20% of the edges in a data flow and aims to predict the masked edges. NA [19] involves masking a certain portion of edges of connecting data flow nodes and code tokens and aims to predict the masked nodes. TMLM [25] masks paths in the AST input on the encoder side and tokens in the code sequence input on the decoder side, then predicts the token of the masked code. NOP [25] is to exchange the positions of some nodes in the path, and distinguish whether the order of nodes in the AST is correct or not.

[2]https://github.com/Unidata/thredds/blob/d2d68f9eee87f345625211324d71d5dc3e162ee1/cdm/src/main/java/thredds/client/catalog/Property.java#L56-L63
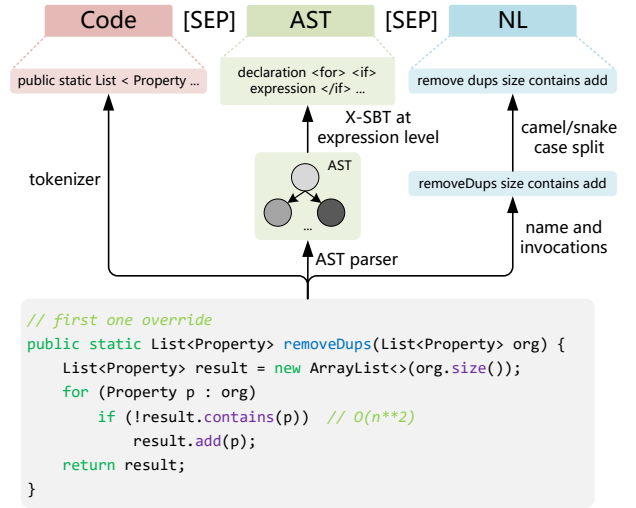
Table 1: Pre-training models in NLP and for source code.

Domain	Models	Modules				Objectives								Inputs		
		LSTM	Encoder	Decoder	Encoder-Decoder	Forward LM	Backward LM	Masked LM	NSP	Permutation LM	RTD	-	-	NL	Code	Structure
NLP	ELMo (2018)	✓				✓	✓							✓		
	BERT (2019)		✓					✓	✓					✓		
	XLNet (2019)		✓							✓				✓		
	RoBERTa (2019)		✓					✓						✓		
	ELECTRA (2019)		✓					✓			✓			✓		
	GPT-2 (2019)			✓		✓								✓		
	T5 (2020)				✓			✓						✓		
	BART (2020)				✓			✓						✓		
Code	SCEMLo (2020)	✓				✓	✓								✓	
	CuBERT (2019)		✓					✓	✓						✓	
	C-BERT (2020)		✓					✓	✓						✓	
	IntelliCode (2020)			✓				✓							✓	
	CodeBERT (2020)		✓					✓			✓			✓	✓	
	GraphCodeBERT (2020)		✓					✓				EP	NA	✓	✓	Data Flow
	CugLM (2020)		✓			✓		✓	✓						✓	
	T5-learning (2021)				✓			✓						✓	✓	
	TreeBERT (2021)				✓							TMLM	NOP		✓	AST

(a) When SPT-Code is used for classification, the inputs of the encoder and decoder are identical, and the output of the decoder at the last time step is used as the label for the classification.

(b) When SPT-Code is used for generation, the whole procedure is the same as the regular sequence-to-sequence model.

Figure 1: SPT-Code for classification and generation.

Figure 2: The input of a real world Java code snippet. Due to space constraints, we abbreviated the node names in the AST sequence. "NL" denotes the natural language input.

3.2.1 Code Tokens. As we can see from Figure 2, the first part of the input is the code token sequence of a method. We use a lexical analyzer to tokenize the source code and then obtain the tokens $C = \{c_1, c_2, \ldots, c_l\}$, where l is the number of code tokens. Specifically, we use the Python standard library[3] to tokenize Python codes. For languages such as Java, JavaScript, PHP and Go, we use the Python binding[4] of ANTLR 4[5] to get the code tokens. The Ruby source code is tokenized by the calling of a Ruby binary program. The source codes of other programming languages are tokenized by the NLTK tokenizer[6].

3.2.2 Linearized AST. To represent the second part of the input, i.e., the structural information of the code, we convert an AST into a specially formatted sequence by traversing it, and call the result

of this converting a *linearized* AST. We first employ an AST parser[7] to get the corresponding AST, then use a traversal method to parse the AST into a sequence.

Instead of using the original SBT (please refer to [20, 21] for more details), which has been shown to be more effective than classical traversal methods (i.e., pre-order traversal) but tend to produce excessively long sequences that are on average more than three times the length of the code, we propose a simplified version of SBT called X-SBT (XML-like SBT) to traverse ASTs. X-SBT can reduce the length of the resulting sequence of traversals by more than half, Figure 3 shows a comparison of SBT and X-SBT. It can be seen that when traversing AST, SBT takes two tokens, "(" and the node name, as the starting flag of a certain AST node, and takes ")" and the name as the ending flag. We make one observation: for non-terminal nodes (i.e., nodes which are not leaves), we can replace the starting flag with one token in an XML-like form, similarly for

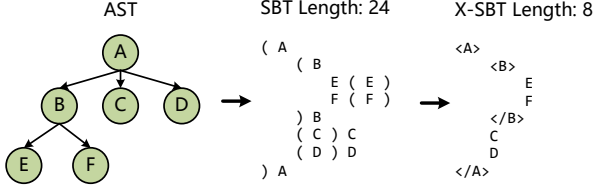---

[3]https://docs.python.org/3.8/library/tokenize.html
[4]https://pypi.org/project/antlr4-python3-runtime/
[5]https://github.com/antlr/antlr4
[6]https://www.nltk.org/api/nltk.tokenize.html

[7]https://tree-sitter.github.io/tree-sitter/

Figure 3: An example of the SBT and X-SBT, traversal sequences are formatted and indented for a better illustration.

the ending flag. For terminal nodes, i.e., leaf nodes, we can further merge the starting and ending tokens into one token. Therefore, it is easy to prove that X-SBT shortens the length by more than half.

However, X-SBT sequences are still long. To further shorten X-SBT sequences, we make X-SBT traverse only the nodes at or above the *expression* level in the AST. This will also reduce redundancy. Commonly, the AST contains both lexical and syntactic information, where the lexical information is already represented by the first part of the input (i.e., code tokens). Since the lexical information in the AST is concentrated on the terminal nodes, we only keep the nodes at or above the *expression* level, so that it contains only syntactic information. Figure 4 shows the nodes in the AST that will be
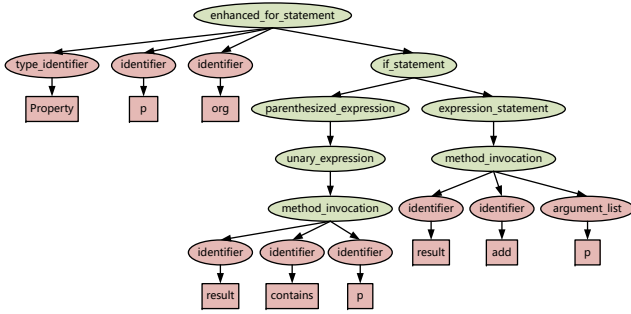

Figure 4: The AST of the "for" statement block of the Java code snippet in Figure 2. Non-terminal nodes are ellipses and terminal nodes are represented as rectangles. The green nodes will be traversed by X-SBT at the expression level, while the red ones will not.

traversed by the X-SBT at the *expression* level. Obviously, it traverses only one subtree of the AST, and consequently, it can further reduce the length of the sequence by ignoring some fine-grained (lexical) information that is already present in the code tokens. To conclude, we traverse the AST using X-SBT at the *expression* level to obtain the tokens $A = \{a_1, a_2, \ldots, a_m\}$, where m is the length of the sequence.

3.2.3 Natural Language. For extracting natural language information from the code only, we derive the method name and API call sequence of the code[8]. For example, the tokens extracted from the code snippet in Figure 2 are `removeDups`, `size`, `contains`, `add`. We further split each token of the form *CamelCase* and *snake_case*, so `removeDups` is split into `remove` and `dups`. Then we take the

---
[8]The reason we do not use documentation in code, such as docstring and in-line comments, is that documentation is not always available as we mentioned in Section 1.

resulting linear sequence of tokens, $N = \{n_1, n_2, \ldots, n_p\}$, as our natural language input, where p is the number of tokens.

After completing the construction of the three input parts, we concatenate them and use a special symbol, i.e., "[SEP]", to separate the three inputs. Therefore, the input is represented as

$$Input = c_1, \ldots, c_l, [SEP], a_2, \ldots, a_m, [SEP], n_1, \ldots, n_p \quad (1)$$

3.3 Pre-Training Tasks

In this section, we introduce the three tasks in the order they are used for (sequential) pre-training.

3.3.1 Code-AST Prediction. The first pre-training task is Code-AST Prediction (CAP). Since we add structural information to the input, in CAP, we expect to have the model learn such information about the structure represented by X-SBT. So we introduce this binarized task that can be trivially generated from any code. Specifically, when we construct the input representation, namely $Input = C,[SEP],A,[SEP],N$, 50% of the time A is the actual AST sequence corresponding to C (labeled as IsAST), and 50% of the time it is a random AST sequence from the dataset (labeled as NotAST). As we show in Figure 1a, "label" is used for Code-AST Prediction. Note that CAP is closely similar to the next sentence prediction task [13].

3.3.2 MASS. Since SPT-Code is based on the encoder-decoder architecture, we expect to pre-train SPT-Code in a sequence-to-sequence style. Therefore, we then adopt MASS, which seeks to reconstruct a sentence fragment given the remaining part of the sentence in the encoder-decoder framework. We employ a modified version of MASS as one of our pre-training tasks, with the intention of training the model to understand, infer and generate code sequences.

Given an input $Input = C,[SEP],A,[SEP],N$, we denote $C_{\text{origin}}^{\backslash u:v}$ as a modified version of C where its fragment from position u to v is masked, that is,

$$C_{\text{origin}}^{\backslash u:v} = \{c_0, \ldots, c_{u-1}, [MASK], \ldots, [MASK], c_{v+1}, \ldots, c_l\} \quad (2)$$

where $0 < u < v < l$. In contrast, $C^{u:v}$ denotes the fragment of C from u to v. In our work, we merge a number of consecutive "[MASK]" symbols into one, then $C_{\text{origin}}^{\backslash u:v}$ is replaced by $C^{\backslash u:v}$:

$$C^{\backslash u:v} = \{c_0, \ldots, c_{u-1}, [MASK], c_{v+1}, \ldots, c_l\} \quad (3)$$

We utilize this modified version of MASS to pre-train our sequence to sequence model by predicting the fragment $C^{u:v}$ taking the input $Input = C^{\backslash u:v},[SEP],A,[SEP],N$. Then, $k = u - v + 1$ is the number of the masked consecutive tokens, and we follow Song et al. [49] and set k=50% of l to achieve the best performance.

3.3.3 Method Name Generation. With the last pre-training task, we would expect the model to learn information such as the intent and functionality of the code. The method name, which is present in every method, can be seen as an extremely concise summary of the method. Xie et al. [59] analyze the method name and code summary in a Java dataset built by Leclair and McMillan [33], and find that on average 50.6% of the words in method names appear in the corresponding summaries, and 21.3% of the words in the summaries appear in the corresponding method names. For about

Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo

20% of the methods, all the words in the method names appear in the corresponding summaries. Therefore, they improve code summarization performance successfully by predicting method names in advance.

Therefore, we exploit method names in this pre-training task, which we call Method Name Generation (MNG). Given the model input $Input = C,[SEP],A,[SEP],N$, we denote $name = c_i$ as the method name in C, where i is the index of the method name token in C. From Section 3.2.3, we learn that the split $name = c_i$ also appears in N, so we remove the tokens from N that are derived from the method name split. Assuming that the method name is split into s subtokens, the final input of MNG will be as follows:

$$
\begin{aligned}
Input_{MNG} = & c_0, \ldots, c_{i-1}, [MASK], c_{i+1}, \ldots, c_l, \\
& [SEP], a_1, a_2, \ldots, a_m, \\
& [SEP], n_{s+1}, n_{s+2}, \ldots, n_p
\end{aligned} \tag{4}
$$

Then we make the decoder try to generate the split method name, i.e., $n_1, n_2, \ldots, n_s$.

3.4 Fine-Tuning

We group all downstream tasks into two categories: classification and generation. For classification tasks, we use the setting shown in Figure 1a; otherwise, we use the setting in Figure 1b. For each task, we simply plug in the task-specific inputs and outputs into SPT-Code and fine-tune all the parameters end-to-end. For example, in code search, which we cast as a classification task, when obtaining a representation vector of a natural language query, we take only the natural language as input and do not consider the first two parts of the input, i.e., code and AST. For code summarization, which we cast as a generation task, we only include the natural language summary in the output. We will describe the task-specific details in the corresponding subsections of Section 4.

4 EXPERIMENT

In this section, we first introduce the pre-training data and settings (Section 4.1) and how we fine-tune on the downstream tasks (Section 4.2). Then, we answer four research questions (Section 4.3). Finally, we manually evaluate SPT-Code through quantitative and qualitative analysis.

4.1 Pre-Training

4.1.1 Dataset. The dataset we use for pre-training SPT-Code is CodeSearchNet [24], which has also been used to pre-train Code-BERT [16], GraphCodeBERT [19] and T5-learning [39]. The Code-SearchNet corpus is programmatically obtained by scraping open-source repositories and pairing individual functions with their (processed) documentation as natural language annotation. It includes more than 6.4M codes from 6 programming language, i.e., Java, Python, JavaScript, PHP, Go and Ruby. The data statistics are shown in Table 2.

Since our input can be extracted from completely unlabeled data, we can make use of all the 6.4M data instances in CodeSearch-Net. However, CodeBERT and GraphCodeBERT use labels (i.e., the documentation) in the input for training both code and natural language, so they are both pre-trained on all the data in the "w/ documentation" column of the "All" row, which we named $CSN_{w/doc}$.

Table 2: Pre-Training Dataset Size Statistics

Language	# Methods/Function	
	w/ documentation	All
Java	542,991	1,569,889
Python	503,502	1,156,085
JavaScript	157,988	1,857,835
PHP	717,313	977,821
GO	347,789	726,768
Ruby	57,393	164,048
All	2,326,976	6,452,446

T5-learning, inputs the code and the documentation into the model separately when pre-training, so it resembles SPT-Code in the sense that it does not utilize the code-document correspondence. Nevertheless, the publicly available implementation of T5-learning uses only the data in the "All" column of "Java" row, named CSN_{Java}.

4.1.2 Settings. The three pre-training tasks of SPT-Code are performed sequentially. We tried different orders and different numbers of epoch for these tasks and found that better results can be achieved by first pre-training CAP for 10 epochs, then MASS for 30 epochs, and eventually MNG for 30 epochs. All three tasks use cross-entropy as the loss function. We use AdamW [38] as our optimizer, the initial learning rate is 5e-5 and the warmup step is set as 2000. We pre-train SPT-Code on 4×NIVDIA A100s[9] with a total batch size of 256.

We use Byte-Pair Encoding (BPE) [47] to tokenize the code and the natural language (after CamelCase and snake_case splitting), and use regular word tokenizer to tokenize X-SBT sequences. The tokenizer is built upon the whole pre-training data, and will be directly used on each downstream tasks without any modification.

4.2 Fine-Tuning on Downstream Tasks

In this subsection, we detail the process of fine-tuning the pre-trained SPT-Code on the five downstream tasks. For each downstream task, we give a brief introduction, tell the datasets for fine-tuning, present the compared baselines, and eventually illustrate the metrics for evaluating.

4.2.1 Code Summarization. Code summarization, also known as the process of code summary or code comment generation, is the task of automatically generating a natural language description for a piece of source code that summarizes the overall actions of the code snippet accurately, adequately, and concisely [50]. Work in this area of research has generally focused on generating a (typically short) natural language comment from a given method.

Datasets. In addition to CodeSearchNet, we use two widely used classical datasets that are collected from open source repositories in GitHub, i.e., the Java dataset (JCSD) [22] and the Python dataset (PCSD) [40]. As for JCSD, the comment of each method is the first

---

[9]The GPUs are provided by Alibaba Group, which employ EFLOPS [14] and Alibaba Collective Communication Library (ACCL) [15] techniques. EFLOPS is a high performance distributed training system that can achieve near-linear scalability of overall throughput, and ACCL brings the performant efficiency of EFLOPS architecture to general cluster systems and Cloud scenarios, which is able to achieve fully non-congested communication.

sentence extracted from its Javadoc, and the comment of a method in PCSD is its docstring.

Baselines. We adopt CodeBERT [16], GraphCodeBERT [19], CugLM [37], T5-learning [39] and TreeBERT [25] as baselines for all datasets[10]. For JCSD, PCSD and CodeSearchNet's Java/Python datasets, we adopt NerualCodeSum [2], which employs Relative Position [48] and Copy Attention [46] upon vanilla Transformer [54], as another baseline[11]. Besides, we use three more baselines for JCSD and PCSD, namely, DeepCom [20], Rencos [62], and SiT [58], which are widely compared or recently proposed RNN or transformer-based models dedicated for code summarization.

Metrics. We use BLEU (B.) [42], METEOR (M.) [7] and ROUGE-L (R.L) [36], which are widely used in fields such as machine translation and text summarization, to measure the similarity between the sentences generated by the model and the goal sentences.

4.2.2 Code completion. Code completion is to generate code given its surrounding code as context. We fine-tune SPT-Code on a task called *any-code* completion [5] here. Unlike *restricted* completion where the target code contains only primitive types (e.g., `int` and `string`) and excludes user-defined functions, *any-code* completion aims to generate code in a general-purpose programming language without any restriction on its vocabulary or structure. Specifically, consider a program $\mathcal{P}$ and some part of the program $p \in \mathcal{P}$, any-code completion makes the model to predict p given the rest of the program $\mathcal{P}^- = \mathcal{P}/p$.

Datasets. The dataset we use for any-code completion is the Java dataset provided by Alon et al. [5]. It is based on the Java-small dataset [4], which contains the least code duplication [3]. Alon et al. create any-code completion examples by selecting every expression larger than a single AST node as the target p, using the reminder of the method as the context $\mathcal{P}$. They also filter many methods to clean the dataset and make the task harder. The resulting dataset contains 1.3M/10k/20k train/dev/test examples.

Baselines. Besides CodeBERT, GraphCodeBERT, CugLM, T5-learning and TreeBERT, some specific methods for this task are also compared. Code2seq [4] represents a code snippet as the set of compositional paths in its AST and uses attention to select the relevant paths while decoding. Transformer w/ copy uses the implementation of Open-NMT [30] with a copy mechanism [17]. Seq2tree w/ copy [1] learns to generate the linearized, subtokenized target AST for code completion. SLM [5] leverages joint modeling of an AST and its missing subtree using a structural language model, which estimates the probability of the program's AST by decomposing it into a product of conditional probabilities over its nodes.

Metrics. Following Alon et al. [5], we use exact match accuracy at 1 (Acc@1) and 5 (Acc@5) for evaluation. An exact match is counted if and only if the sentence generated by the model is identical to the goal sentence (ignoring cases and whitespaces). If we let the model generate k candidate sentences with the highest probability and if an exact match is counted while any one of the candidates matches the goal sentence exactly, the ultimate exact match accuracy is Acc@k (e.g., if only one candidate, it is Acc@1).

---

[10] We resize all the pre-trained models here and in all the following downstream tasks identical to ours.
[11] We have tried to run it as Java for other languages in CodeSearchNet but got completely nonsensical output.

4.2.3 Bug fixing. Bug fixing, aka. code repair or code refinement, aims to automatically fix bugs in the code. It can help software developers locate and fix bugs and thus save a lot of time [26, 53].

Datasets. We use the dataset collected by Tufano et al. [53], who extract method-level pairs of buggy and corresponding fixed code named BFPs (bug-fix pairs) from bug-fixing commits in thousands of GitHub Java repositories. Each BFP is composed of a tuple $BFP = <m_b, m_f>$, where m_b represents a buggy code component, m_f represents the corresponding fixed code. Based on the code size, Tufano et al. provide two datasets: BFP_{small} and BFP_{medium}, with the former having a code length below 50 and the latter having a code length between 50 and 100.

Baselines. The original method proposed by the dataset creator [53] is marked as *Tufano et al.* S2S+COPYSAPN [41] is an extension of seq2seq models which can copy entire spans of the input to the output in one step and reduce the number of decisions required during inference. CodeBERT, GraphCodeBERT, CugLM, T5-learning and TreeBERT are also used as baselines.

Metrics. We report Acc (=Acc@1) and BLEU on both datasets. Both m_b and m_f are abstracted (please refer to [53] for more details) before being fed to the model, and we do not translate the abstracted code predicted by the model back into source code before evaluation using the metrics because the result before and after translation are the same.

4.2.4 Code Translation. Code translation is important for migrating legacy code in one language into an ecosystem built in a different language.

Datasets. Following Chen et al. [10] and Guo et al. [19], we conduct our experiments on a dataset containing several open-source projects, which have both a Java and a C# implementation.

Baselines. Baselines are Naive (i.e., directly copying the source code as the translation result), vanilla Transformer [54], CodeBERT, GraphCodeBERT, CugLM, T5-learning and TreeBERT.

Metrics. We use Acc and BLEU as metrics.

4.2.5 Code Search. Code search aims to find the code snippet that most closely resembles the semantics of the given natural language query statement from a set of codes, which is named *codebase* in this task. It is a good choice for testing the performance of SPT-Code in classification mode. For each code-query pair (c, q), we compute the representation vectors of c and q, i.e., V_c and V_q respectively. Then, we randomly take another query statement from the dataset as a negative sample, denoted as $q-$, whose representation vector is V_{q-}. To ensure the effectiveness of the negative samples, we restrict the BLEU score [42] between $q-$ and q to be lower than 0.3. The training loss is:

$$\mathcal{L}_{search} = \sum_{(c,q,q-)\in D} \max(0, \varepsilon - \cos(V_c, V_q) + \cos(V_c, V_{q-})) \quad (5)$$

$$\cos(V_c, V_q) = \frac{V_c \cdot V_q}{\|V_c\|\|V_q\|} \quad (6)$$

where D is the dataset, ε is a fixed value of 0.05 (following Gu et al. [18]), and cos denotes the calculation of cosine similarity. In the evaluation, we first get the representation vectors of all codes in codebase as candidates. Then, for each vector of query statements, we select a few code representation vectors from the candidates

that are closest to the vector of query statements. Finally, the metric is calculated.

Datasets. We use CodeSearchNet [24].

Baselines. We use CNN [29] (i.e., 1D convolutional neural network), Bi-GRU [11], and Transformer (i.e., vanilla Transformer [54]) as baselines in addition to CodeBERT and GraphCodeBERT.

Metrics. We use Mean Reciprocal Rank (MRR) for evaluation. MRR is a statistic measure for evaluating search algorithms. The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer: 1 for first place, 1/2 for second place. Therefore, the mean reciprocal rank is the average of the reciprocal ranks of results for a sample of queries Q:

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i} \qquad (7)$$

4.3 Evaluation

We evaluate SPT-Code by answering four research questions.

RQ1: How effective is SPT-Code compared with the state-of-the-art baselines and other code pre-trained models on five downstream tasks? We conduct experiments on five downstream tasks introduced in the previous subsection. The results on classical and CodeSearchNet code summarization are shown in Tables 3 and Table 4, respectively.

Table 3: Results on classical code summarization.

Methods	JCSD			PCSD		
	B.	M.	R.L	B.	M.	R.L
DeepCom	37.5	22.0	51.2	20.4	9.5	36.8
Rencos	36.9	26.5	51.2	28.9	20.4	45.0
NerualCodeSum	44.5	26.4	54.7	32.3	19.7	46.8
SiT	45.3	27.3	55.0	33.8	20.7	48.2
CodeBERT	45.2	26.2	54.7	33.3	21.6	49.2
GraphCodeBERT	46.0	26.5	56.5	33.9	22.1	50.4
CugLM	46.1	26.3	55.8	34.3	21.6	49.7
T5-learning	44.2	26.5	53.9	34.1	22.6	50.1
TreeBERT	47.9	27.2	56.6	34.7	23.0	50.5
SPT-Code	**49.1**	**32.4**	**58.2**	**36.1**	**26.9**	**52.0**

Tables 5-8 show the results of code completion, bug fixing, code translation and code search, respectively.

First of all, we can see that whether the baselines are dedicated to a specific downstream task or pre-trained models, SPT-Code clearly outperforms them in the vast majority of cases, including code summarization, code completion, bug fixing on BFP$_{medium}$, and code translation. This is because during the pre-training process, SPT-Code learns the representation of codes from a large amount of data, as well as the connection between codes and structural and natural language information. In addition, the model is enhanced to generate code and natural language sequences by pre-training.

As for bug fixing on BFP$_{small}$, the accuracy of SPT-Code is slightly lower than that of S2S+COPYSPAN, which achieves the best accuracy on BFP$_{small}$. The reason we believe is two-fold. The first is the dataset, where the lengths of the code are up to 50, which allows RNN-based S2S+COPYSPAN to adequately cope with inputs of these lengths. Another reason is that the mechanism proposed

by S2S+COPYSPAN to copy the entire span of the input is well suited for a task like bug fixing, where only individual modifications are made on the input. However, in the next RQ, we will show that SPT-Code still outperforms S2S+COPYSPAN on BFP$_{small}$ after removing MNG from the pre-training tasks.

Finally, we can learn that SPT-Code performs comparably to GraphCodeBERT on code search, which suggests that although we designed our model to perform better on generation tasks, it does not come at the cost of the performance on classification tasks.

RQ2: How do the three pre-training tasks, as well as the AST and the natural language input contribute to the model's performance on the different downstream tasks? In order to find the impact of each component to the overall performance of SPT-Code, we conduct ablation study on all downstream tasks. We remove each/all pre-training task, and AST or/and natural language part from the input, respectively. Owing to space limitations, for code summarization, we only show results on Java and Python in CodeSearchNet[12]; and for code search, we only show results on Java and Go[13]. The results are shown in the first three groups of Table 9.

The first thing we notice is that when we train from scratch (i.e., remove all pre-training tasks) or remove the AST and natural language from the input (i.e., only input code tokens), the performance of SPT-Code consistently drops considerably. Second, when we remove each of the three pre-training tasks, the results decreased in most cases, particularly in code summarization, code translation and code search. Of these, CAP is most useful for code search, MASS is most helpful for code translation, and removing MNG has the greatest impact on code summarization.

Interestingly, we find that for code completion and bug fixing, SPT-Code's performance w.r.t. accuracy improves instead when MNG is removed. This is understandable. On the one hand, code completion and bug fixing are tasks where the input and the output are both code, as is MASS, while the output of the MNG task is natural language, and thus the ability to generate natural language trained by MNG is not fully reflected in these two tasks. On the other hand, regarding code completion, MASS can be seen as totally unrestricted any-code completion. Both predict a piece of code based on its context, with the difference that in any-code completion, the piece of code is restricted to be an entire expression, while in MASS, the piece of code is selected completely at random. Therefore, fine-tuning code completion directly after MASS, i.e., removing the MNG, yields a higher result.

When we remove either the AST or the natural language from the input, the results of the model drop, indicating that they both help improve performance. In addition, for code summarization, bug fixing and code search, the results are lower when only natural language is removed, which indicates that in comparison, natural language helps these three tasks more than ASTs do. On the contrary, AST is more helpful for code completion and code translation.

Through ablation, we find that different pre-training tasks show different degrees of influence on the downstream tasks, As a result, appropriate trade-offs of pre-training tasks for different downstream tasks can help the model achieve better performance. ASTs

---

[12]Java and Python represent static and dynamic languages, respectively.
[13]The results for both are above and below GraphCodeBERT, respectively.

Table 4: Results on CodeSearchNet code summarization.

Methods	Java			Python			JavaScript			PHP			Go			Ruby		
	B.	M.	R.L	B.	M.	R.L	B.	M.	R.L	B.	M.	R.L	B.	M.	R.L	B.	M.	R.L
NeuralCodeSum	10.7	12.9	25.9	9.4	4.9	17.0	-	-	-	-	-	-	-	-	-	-	-	-
CodeBERT	13.6	16.4	32.2	10.7	6.0	21.0	10.0	10.5	22.2	20.1	19.3	42.7	16.0	11.7	30.4	8.3	7.5	17.2
GraphCodeBERT	14.5	17.8	33.5	11.0	7.4	22.1	10.9	11.4	24.9	20.0	20.2	42.8	16.5	12.5	30.1	8.3	8.1	18.0
CugLM	13.2	16.1	31.7	11.4	7.5	22.0	10.0	10.4	21.8	17.8	18.2	41.9	17.0	12.7	29.7	7.4	6.8	15.3
T5-learning	13.5	16.8	34.2	9.7	5.5	16.3	9.0	9.2	19.6	18.4	17.9	35.6	15.3	11.0	25.9	7.8	6.7	15.5
TreeBERT	13.8	17.1	34.3	11.1	7.2	21.6	10.3	10.5	22.0	18.0	19.1	42.4	17.1	12.8	30.1	7.4	6.9	15.5
SPT-Code	**16.8**	**20.6**	**36.3**	**12.8**	**14.2**	**27.1**	**12.8**	**17.2**	**28.8**	**20.4**	**20.6**	**43.7**	**18.8**	**16.6**	**38.1**	**8.4**	**11.1**	**20.0**

Table 5: Results on any-code completion.

Methods	Acc@1	Acc@5
code2seq	10.44	15.47
Transformer w/ copy	16.68	24.12
seq2tree w/ copy	16.46	22.89
SLM	18.00	24.77
CodeBERT	17.23	24.79
GraphCodeBERT	18.21	25.00
CugLM	18.43	25.51
T5-learning	16.03	23.78
TreeBERT	17.17	24.73
SPT-Code	**19.09**	**26.57**

Table 6: Results on bug fixing.

Methods	BFP_{small}		BFP_{medium}	
	Acc	BLEU	Acc	BLEU
Tufano et al.	9.27	-	3.21	-
S2S+COPYSPAN	**17.6**	-	7.9	-
CodeBERT	6.23	61.47	2.43	68.54
GraphCodeBERT	7.49	63.40	3.16	67.33
CugLM	14.55	67.24	7.84	71.23
T5-learning	12.01	65.02	4.51	60.89
TreeBERT	13.15	65.74	7.61	70.67
SPT-Code	17.54	**75.10**	**10.86**	**87.88**

Table 7: Results on code translation.

Methods	Java → C#		C# → Java	
	Acc	BLEU	Acc	BLEU
Naive	00.0	18.52	00.0	18.66
Transformer	33.4	56.05	38.8	50.84
CodeBERT	58.8	79.34	57.3	71.10
GraphCodeBERT	59.3	80.33	58.1	72.39
CugLM	61.6	82.72	59.5	74.89
T5-learning	54.1	81.35	48.4	77.10
TreeBERT	60.0	80.92	57.8	77.15
SPT-Code	**64.07**	**90.34**	**60.29**	**86.10**

and natural language both have a positive impact on the performance of the model, regardless of the downstream task.

Table 8: Results on code search.

| Methods	Java	Python	JS	PHP	Go	Ruby
CNN	0.262	0.241	0.224	0.261	0.676	0.274
Bi-GRU	0.312	0.298	0.195	0.340	0.691	0.214
Transformer	0.404	0.399	0.289	0.430	0.729	0.278
CodeBERT	0.673	0.670	0.618	0.624	0.879	0.674
GraphCodeBERT	0.690	0.692	**0.643**	0.647	**0.896**	**0.701**
SPT-Code	**0.700**	**0.699**	0.641	**0.651**	0.895	**0.701**

RQ3: Is the ability of utilizing more unlabeled data an advantage of SPT-Code? As we know, CodeBERT, GraphCodeBERT and T5-learning are pre-trained on a subset of CodeSearchNet, whereas we can pre-train on the entire dataset. Therefore there is a question of whether the ability to utilize more data for pre-training also gives SPT-Code an unfair advantage. To ensure a fair comparison, we therefore pre-train SPT-Code only on the same data as CodeBERT and GraphCodeBERT (i.e., $CSN_{w/\,doc}$), and also on the same data as T5-learning (i.e., CSN_{Java}). Notice, however, that here only the code from the labeled data is utilized and not the labels, even on the $CSN_{w/\,doc}$ where all samples have labels. So in this case, although we are using the data set of the same size, it actually utilize less information than they do. The results are displayed in the last group of Table 9.

Comparing the results of SPT-Code with those of "$CSN_{w/\,doc}$" and "CSN_{Java}" in Table 9, we find that there is a significant performance decrease when the data used for pre-training is shrunk from 6.4M to 2.3M or 1.5M. However, considering the other pre-training models, it still maintains an advantage or is at a comparable level in performance. Therefore, we can conclude that SPT-Code is superior given the same amount of pre-trained data. In addition, the ability to use more unlabeled pre-training data can help SPT-Code achieve higher performance. This ensures that SPT-Code has a very advantageous scalability at the data level compared to the other pre-training models for source code.

RQ4: How would the size of the training data for fine-tuning SPT-Code influence its performance on downstream tasks? To answer this question, we plot learning curves by varying the amount of (task-specific) training data used to fine-tune SPT-Code. Owing to space limitations, we only report results of code summarization on Java in CodeSearchNet, and bug fixing on BFP_{medium}. We select $k/10$ data from the training set each time for training, and then test on the same entire testing set. The results are shown in Figure 5.

Table 9: Ablation study on downstream tasks.

| Methods | Summarization | | | | | | Completion | | Bug fixing | | | | Translation | | | | Search | |
| | Java | | | Python | | | Acc@1 | Acc@5 | BFP$_{small}$ | | BFP$_{medium}$ | | Java → C# | | C# → Java | | Java | Go |
	B.	M.	R.L	B.	M.	R.L			Acc	BLEU	Acc	BLEU	Acc	BLEU	Acc	BLEU		
SPT-Code	**16.79**	**20.55**	**36.34**	**12.77**	**14.16**	**27.10**	19.09	**26.57**	17.54	**75.10**	10.86	87.77	**64.07**	**90.34**	**60.29**	**86.10**	**0.700**	**0.895**
-w/o CAP	15.85	20.33	36.30	12.26	14.16	26.57	18.42	25.63	17.20	73.54	10.06	87.26	57.25	88.80	54.40	84.66	0.668	0.872
-w/o MASS	15.75	20.19	35.70	12.16	13.48	26.00	17.42	23.71	16.30	72.71	9.65	86.10	53.93	82.09	50.55	82.32	0.686	0.889
-w/o MNG	15.45	19.86	35.65	12.01	13.81	25.98	**19.71**	26.10	**18.11**	73.09	**11.96**	**87.79**	58.80	88.79	57.16	85.19	0.673	0.875
-w/o all	13.54	17.72	33.31	11.90	12.74	25.23	13.49	19.31	14.28	72.89	8.27	85.30	49.19	79.28	47.49	78.69	0.656	0.855
-w/o AST	16.43	20.14	36.19	11.87	13.80	25.74	18.53	25.78	16.35	72.85	10.61	87.52	55.81	87.79	56.17	82.66	0.692	0.885
-w/o NL	15.68	20.15	35.86	11.96	13.56	25.55	18.47	26.09	16.12	72.70	10.50	87.57	56.29	87.83	56.92	83.56	0.679	0.865
-only code	15.35	19.97	35.63	10.25	12.74	23.90	18.33	25.88	16.01	72.55	10.19	86.96	54.13	87.34	54.59	83.81	0.669	0.864
-CSN$_{w/ doc}$	15.41	19.78	35.74	12.59	13.56	26.49	17.89	23.80	16.14	74.02	10.41	87.58	60.53	88.78	58.28	84.09	0.673	0.878
-CSN$_{Java}$	15.67	19.72	35.22	12.40	12.60	25.24	17.76	23.54	16.10	74.51	10.18	86.96	59.43	86.55	58.11	83.07	0.678	0.866
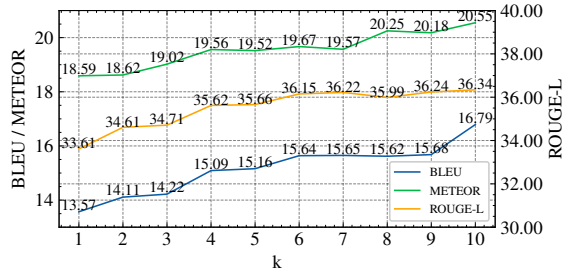
(a) Results on Java CodeSearchNet code summarization.
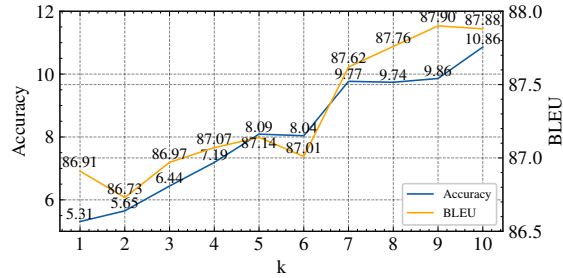
(b) Results on BFP$_{medium}$ bug fixing.

Figure 5: Results when k goes from 1 to 10.

It is easy to see that the performance shows a decreasing trend in all evaluation metrics as k decreases (i.e., the size of the training set decreases). It proves that even for pre-trained models, the size of the training set plays a key role in the performance as well. Meanwhile, considering Table 4, we see that for code summarization, the results of SPT-Code are comparable to that of CodeBERT when the training set size is reduced to 1/10, and the performance is similar to or even higher than GraphCodeBERT when it is reduced to 2/10. Further, recalling Table 6, we can see that when the training set of BFP$_{medium}$ is reduced to 1/10 of the original size, SPT-Code's results are still higher than other pre-trained models, and when it is reduced to half of the original size, SPT-Code's performance can reach the performance of the best baseline model, i.e., S2S+COPYSPAN.

The conclusion is that although there is an inevitable drop in SPT-Code's performance as the size of the training data decreases

during fine-tuning, the performance of SPT-Code is comparable to that of the other models when the data used for fine-tuning SPT-Code is reduced to a very small size. This also implies the robustness of SPT-Code.

4.4 Quantitative Analysis

We collect the output of SPT-Code and baselines for a wide range of test samples on each downstream task. Then we invited five participants, all of whom are graduate students in software engineering who are themselves not authors of this paper. For each downstream task, the five students are randomly assigned to the same number of samples. If there are multiple datasets for that downstream task, the samples of multiple datasets are the same. For downstream tasks that use Acc as a metric, such as code completion, we focus on the similarity between incorrect and correct answers, and the extent to which the incorrect answer can be helpful when all models fail. The results are shown in Table 10.

Table 10: Results of quantitative analysis.

Dowstream Tasks	# Sample	# Better	# Comparable	# Worse
Code Summarization	800	328	368	104
Code Completion	600	312	233	55
Bug Fixing	200	89	75	36
Code Translation	200	83	65	52
Code Search	300	61	185	54

4.5 Qualitative Analysis

By asking the participants in the previous subsection and browsing the output of each model by ourselves, we found that in comparison, SPT-Code can capture the semantic information of identifiers within code more accurately than other pre-trained models, and it can capture the semantic information of code segment globally instead of limiting to a certain region. The extracted code semantics are relatively more widely and evenly distributed in the code segment.

In the case of code summarization, for example, SPT-Code provides more complete and accurate descriptions of the method's overall functionality. Table 11 shows summaries generated by different models for an example Java method in CodeSearchNet data. We find that the "queue" object is recognized by all of these models. CodeBERT and T5-learning fail to capture the operation conducted

Table 11: Qualitative example of SPT-Code and baselines.

```
void emitLoop() {
    for ( ; ; ) {
        AppendOnlyLinkedArrayList<Object> q;
        q = queue;
        if (q == null) {
            emitting = false;
            return; }
        queue = null;
        q.forEachWhile(this); } }
```

CodeBERT: the queue is being destroyed
GraphCodeBERT: emits the next loop in the queue
T5-learning: this method is called when the queue

SPT-Code: emits all of the elements in this queue
 -w/o AST: emit the linked list
 -w/o NL: emits all entries in the queue
Human Written: loops until all notifications in the queue has been processed

on the queue, i.e., "emits/emit", but GraphCodeBERT and SPT-Code does. Compared to SPT-Code, although GraphCodeBERT also understand the relationship between the operation and the object, it fails to choose appropriate words for describing the operation. The reason may be that the data flow can indeed partially (but not fully) capture code structure information. Another interesting observation that may support this inference is that all models considering structure information generate "in the queue" correctly, i.e., Graph-CodeBERT, SPT-Code, and SPT-Code-w/o NL, and models using AST all generate accurate, readable and smooth summaries. It also implies that data flow is less effectiveness than AST.

5 THREATS TO VALIDITY

Construct Validity. Like many existing code pre-training models, SPT-Code uses CodeSearchNet for pre-training. Since CodeSearch-Net is also used in the evaluation of code summarization and code search, it is possible that the samples from the test sets for these two tasks have already been used for pre-training. This is not fair to methods that have not been pre-trained with CodeSearchNet, such as NeuralCodeSum in CodeSearchNet code summarization. We also recognize the impact on the results of not removing duplicates. However, our decision of not removing replicates is based on two considerations: (1) ensuring fairness of the comparison: the three code pre-training baselines were pre-trained with all data from CodeSearchNet without duplicate removal; (2) SPT-code is not affected by duplicate data: on one hand, downstream tasks that use pre-training dataset are CSN code summarization and code search, while the pre-training tasks designed for SPT-Code do not use the docstring, i.e., in testing, SPT-Code does not have the advantage of generating more accurate summaries because it has seen a piece of the tested code during pre-training, or has the advantage of better searching for code based on natural language. On the other hand, other downstream tasks that do not use CodeSearchNet have little overlap with CSN.

For rigorous consideration, we remove all test sets in CodeSearch-Net and code duplicated with test sets of other downstream tasks[14]. Then re-pre-train and fine-tune the SPT-Code on three downstream tasks, i.e., JCSD, code completion and BFP$_{medium}$ bug fixing. It is

found that the results of SPT-Code decrease very little after removing these duplicates[15]. Moreover, it is still not sure whether the change in results is due to the smaller pre-training dataset (shrunk by about 1/10) or the removal of duplicates.

Internal Validity. It is widely agreed that hyperparameters have a significant impact on the performance of deep learning models, but hyperparameters of SPT-Code are not tuned experimentally and are set empirically. Therefore, other hyperparameter settings may yield better results.

External Validity. Another threat posed by using CodeSearchNet as our pre-training dataset is that CodeSearchNet data is not balanced across six programming languages, which can be seen in Table 2, so our model may not perform the same on different languages, and we cannot guarantee the validity of SPT-Code for programming languages other than these six.

6 CONCLUSION

We presented SPT-Code, a large model for source code based on an encoder-decoder architecture. First, we design three code-specific pre-training tasks to pre-train SPT-Code. Secondly, we propose a new input representation whose is the first method that take into account both natural language and AST form of code, where we also propose a improved version of the AST traversal method, X-SBT. Both our pre-training tasks and input representation allow SPT-Code to be pre-trained on a completely unlabeled dataset. SPT-Code was then fine-tuned on five code-related downstream tasks. Results indicate that fine-tuning SPT-Code enables it to achieve the state-of-the-art performance on five code-related downstream tasks. Ablation experiments reveal that the three pre-training tasks have different degrees of impact on different downstream tasks, and AST and natural language input also helped improve SPT-Code's performance. To facilitate future research, we also make our code and other artifacts publicly available at https://github.com/NougatCA/SPT-Code.

REFERENCES

[1] Roee Aharoni and Yoav Goldberg. 2017. Towards String-To-Tree Neural Machine Translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 132–140.

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4998–5007.

[3] Miltiadis Allamanis. 2019. The Adverse Effects of Code Duplication in Machine Learning Models of Code. In *Proceedings of the 2019 ACM SIGPLAN International*

---

[14]By using tools provided by Allamanis [3], we find 9 in classical code summarization, 2 in BFP$_{medium}$ and code completion, respectively.

[15]Results decrease 0.01 BLEU on JCSD, 0.03 Acc@1 on code completion and 0.07 Acc on BFP$_{medium}$

Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. 143–153.

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations.*

[5] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural Language Models of Code. In *International Conference on Machine Learning.* PMLR, 245–256.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[7] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization.* 65–72.

[8] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. 2020. Exploring Software Naturalness through Neural Language Models. arXiv:2006.12641 [cs.CL]

[9] Chi Chen, Xin Peng, Zhenchang Xing, Jun Sun, Xin Wang, Yifan Zhao, and Wenyun Zhao. 2021. Holistic Combination of Structural and Textual Code Information for Context based API Recommendation. *IEEE Transactions on Software Engineering* (2021).

[10] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree Neural Networks for Program Translation. In *Advances in Neural Information Processing Systems*, Vol. 31. Curran Associates, Inc.

[11] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).* 1724–1734.

[12] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2019. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *International Conference on Learning Representations.*

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers).* 4171–4186.

[14] Jianbo Dong, Zheng Cao, Tao Zhang, Jianxi Ye, Shaochuang Wang, Fei Feng, Li Zhao, Xiaoyong Liu, Liuyihan Song, Liwei Peng, et al. 2020. Eflops: Algorithm and system co-design for a high performance distributed training platform. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 610–622.

[15] Jianbo Dong, Shaochuang Wang, Fei Feng, Zheng Cao, Heng Pan, Lingbo Tang, Pengcheng Li, Hao Li, Qianyuan Ran, Yiqun Guo, et al. 2021. ACCL: Architecting Highly Scalable Distributed Training Systems with Highly-Efficient Collective Communication Library. *IEEE Micro* (2021).

[16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings.* 1536–1547.

[17] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. 2016. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* 1631–1640.

[18] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE).* IEEE, 933–944.

[19] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations, ICLR 2021.*

[20] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC).* IEEE, 200–20010.

[21] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep Code Comment Generation with Hybrid Lexical and Syntactical Information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.

[22] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI 2018.* 2269–2275.

[23] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2020. Control Flow Graph Embedding Based on Multi-Instance Decomposition for Bug Localization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 4223–4230.

[24] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436 [cs.LG]

[25] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. TreeBERT: A tree-based pre-trained model for programming language. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, Vol. 161. PMLR, 54–63.

[26] Magne Jorgensen and Martin Shepperd. 2007. A Systematic Review of Software Development Cost Estimation Studies. *IEEE Transactions on Software Engineering* 33, 1 (2007), 33–53. https://doi.org/10.1109/TSE.2007.256943

[27] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119).* PMLR, 5110–5121.

[28] Rafael Michael Karampatsis and Charles Sutton. 2020. SCELMo: Source Code Embeddings from Language Models. arXiv:2004.13214 [cs.SE]

[29] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014.* ACL, 1746–1751.

[30] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. 2017. OpenNMT: Open-Source Toolkit for Neural Machine Translation. In *Proceedings of ACL 2017, System Demonstrations.* 67–72.

[31] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. In *Proceedings of the 28th International Conference on Program Comprehension.* 184–195.

[32] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE, 795–806.

[33] Alexander LeClair and Collin McMillan. 2019. Recommendations for Datasets for Source Code Summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers).* 3931–3937.

[34] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics.* 7871–7880.

[35] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving Code Summarization with Block-wise Abstract Syntax Tree Splitting. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021.*

[36] Chin-Yew Lin and Eduard Hovy. 2002. Manual and Automatic Evaluation of Summaries. In *Proceedings of the ACL-02 Workshop on Automatic Summarization - Volume 4.* 45–51.

[37] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task Learning based Pre-trained Language Model for Code Completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering.* 473–485.

[38] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations.*

[39] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, 336–347.

[40] Antonio Valerio Miceli-Barone and Rico Sennrich. 2017. A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers).* 314–319.

[41] Sheena Panthaplackel, Miltiadis Allamanis, and Marc Brockschmidt. 2021. Copy That! Editing Sequences by Copying Spans. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 13622–13630.

[42] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics.* 311–318.

[43] Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep Contextualized Word Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers).* 2227–2237.

[44] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[45] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[46] Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* 1073–1083.

[47] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1715–1725.

[48] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-Attention with Relative Position Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. 464–468.

[49] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2019. MASS: Masked Sequence to Sequence Pre-training for Language Generation. In *International Conference on Machine Learning*. PMLR, 5926–5936.

[50] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards Automatically Generating Summary Comments for Java Methods. In *25th IEEE/ACM International Conference on Automated Software Engineering, ASE 2010*. ACM, 43–52.

[51] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode Compose: Code Generation Using Transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[52] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: AI-assisted Code Completion System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2727–2735.

[53] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[55] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S Yu. 2019. Multi-Modal Attention Network Learning for Semantic Source Code Retrieval. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. 13–25.

[56] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[57] Yanlin Wang and Hui Li. 2021. Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 14015–14023.

[58] Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code Summarization with Structure-induced Transformer. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*. Association for Computational Linguistics, 1078–1090.

[59] Rui Xie, Wei Ye, Jinan Sun, and Shikun Zhang. 2021. Exploiting Method Names to Improve Code Summarization: A Deliberation Multi-Task Learning Approach. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021*. IEEE, 138–148.

[60] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. XLNET: Generalized Autoregressive Pretraining for Language Understanding. *Advances in Neural Information Processing Systems* 32 (2019).

[61] Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. 2021. A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021*. IEEE.

[62] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based Neural Source Code Summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1385–1397.

[63] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.